Divide & Conquer Group Exercises – KEY – 50 pts

Adapted from material by Cole Frederick

Please work in groups of 2 or 3 to work the following problems. Use additional paper as needed, and staple the sheets together before turning them in. ONLY TURN IN 1 WORKSHEET/ANSWERS PER TEAM.

1. **4 pts** The stages of a divide and conquer strategy are:

   - Divide the problem into smaller instances of the same problem
   - Conquer the subproblems, solving recursively
   - Combine/Merge solutions to subproblems into a solution for the original problem

2. **5 pts** Recursion is:

   Solving a problem by reducing it to instances of the same problem with a smaller input

3. **6 pts** What are the recurrences for:

   - (a) Merge sort \( T(n) = 2T(n/2) + \Theta(n) \)
   - (b) Binary search \( T(n) = T(n/2) + \Theta(1) \)
   - (c) Build heap \( T(n) = nT(n/2) + \Theta(n) \)

**Maximum subarray problem**

Ex: maximize profit for buying a stock on one day and selling it another. We know about the future, and the data we have is the date and the price of the stock on that date.

4. **5 pts** Brute force method – look at how much we could make between every pair of dates and pick the pair that gives the maximum profit:

   Complexity: \( \binom{n}{2} = \frac{n \times (n-1)}{2} = O(n^2) \)

5. **5 pts** Transform list of prices for each day to a list of daily changes in price. Then we can look for (choose one):

   - any subarray and take its maximum sum
   - the longest subarray with the maximum sum
   - the shortest subarray with the maximum sum

Divide and Conquer Strategy for this problem:

6. **3 pts** Divide the array into \( \underline{2} \) \( \underline{\text{___________}} \) chunks. (How many chunks?)
7. **3 pts** A maximum subarray is either contained completely in **left** \((A[low...mid])\), completely in **right** \((A[mid + 1..high])\), or **crossing the mid-point** \((A[i..j], low \leq i \leq mid \leq j \leq high)\)

8. **4 pts** We can use **divide and conquer/recursion** to find it in the first 2 cases. (What strategy?)

9. **5 pts** For the third case, we can compute the maximum subarray in linear time by finding the starting and ending positions, of such a maximum subarray. What two elements are (by definition) included in this subarray?

\[
\text{mid} + 1 \text{ and mid - 1}
\]

**NOTE:** Credit given for all plausible answers

**Analysis:**

Answer the questions about the pseudo-code that follows.

10. **5 pts** What is the complexity of the \texttt{FIND-MAX-CROSSING-SUBARRAY} procedure?

\(\Theta(n)\)

11. **5 pts** What is the recurrence that describes the \texttt{FIND-MAXIMUM-SUBARRAY} procedure? What is its solution? You can assume that the problem size \(n\) is a power of 2 so that all sub-problem sizes are integers.

\[
T(n) = \Theta(1) \text{ if } n = 1, \text{ } 2T(n/2) + \Theta(n) \text{ if } n > 1, \text{ solution is: } T(n) = \Theta(n \log n)
\]

**FIND-MAXIMUM-SUBARRAY**\((A, \text{low, high})\)

1. \textbf{if} \(\text{high} == \text{low}\)
2. \textbf{return} \((\text{low, high, } A[\text{low}])\)
3. \textbf{else} \(\text{mid} = [\text{(low + high/2)}]\)
4. \(\text{(left-low, left-high, left-sum) =}
      \text{FIND-MAXIMUM-SUBARRAY} (A, \text{low, mid})
5. \(\text{(right-low, right-high, right-sum) =}
      \text{FIND-MAXIMUM-SUBARRAY} (A, \text{mid + 1, high})
6. \(\text{(cross-low, cross-high, cross-sum) =}
      \text{FIND-MAX-CROSSING-SUBARRAY} (A, \text{low, mid, high})
7. \textbf{if} \(\text{left-sum} \geq \text{right-sum} \text{ and left-sum} \geq \text{cross-sum}\)
8. \textbf{return} \((\text{left-low, left-high, left-sum})\)
9. \textbf{elseif} \(\text{right-sum} \geq \text{left-sum} \text{ and right-sum} \geq \text{cross-sum}\)
10. \textbf{return} \((\text{right-low, right-high, right-sum})\)
11. \textbf{else return} \((\text{cross-low, cross-high, cross-sum})\)

**FIND-MAX-CROSSING-SUBARRAY**\((A, \text{low, mid, high})\)

1. \(\text{left-sum} = \infty\)
2. \(\text{sum} = 0\)
3. \textbf{for} \(i = \text{mid down to low}\)
4. \(\text{sum} = \text{sum} + A[i]\)
5. \textbf{if} \(\text{sum} > \text{left-sum}\)
6. \(\text{left-sum} = \text{sum}\)
7. \(\text{max-left} = i\)
8. \(\text{right-sum} = \infty\)
9. \(\text{sum} = 0\)
10. \textbf{for} \(j = \text{mid C 1 to high}\)
11. \(\text{sum} = \text{sum} + A[j]\)
12. \textbf{if} \(\text{sum} > \text{right-sum}\)
13. \(\text{right-sum} = \text{sum}\)
14. \(\text{max-right} = j\)
15. \textbf{return} \((\text{max-left, max-right, left-sum + right-sum})\)
Observations on sub-problem sizes: **12-17 not graded; too much ambiguity and confusion**

We’ve mostly dealt with sub-problems that are around half of the size of the bigger problems. Let’s see what happens if we divide them very differently.

Assume we’re working on an algorithm to find the biggest integer in a Python list. We decide to create sub-problems by taking the first element as one sub-problem, and all the rest as the second sub-problem:

MaximumNum is the maximum of L[0] and MaximumNum(L[1:]) (We use Python slicing here for the rest of the array)

12. **5 pts** What is the recurrence if we create sub-problems this way? Hint: the first split gives us T(n-1) + O(1)

\[ T(n) = \sum_{i=1}^{n} O(1) \]

13. **5 pts** What is the solution of this recurrence?

O(n)

Now assume we decide to try taking chunks of 3 numbers. So MaximumNum is the maximum of L[0:3] and MaximumNum(L[3:])

14. **5 pts** What is the recurrence if we create sub-problems this way? Hint: you have to figure out the maximum of each group of 3 numbers, assume using 2 comparisons, so the first split gives us T(n-3) + O(2)

Assume n is divisible by 3 so we don’t have to deal with floors/ceilings. \( T(n) = \sum_{i=1}^{n/3} O(3) \) – we have to pick one of the 3 and then make 2 comparisons.

15. **5 pts** What is the solution of this recurrence?

O(n)

16. **5 pts** What is the recurrence if we decide to create sub-problems from c numbers? (The first sub-problems are L[0:c] and L[c:]

\[ T(n) = \sum_{i=1}^{n/c} O(c) \] – again we have to pick one of the c and then make c-1 comparisons with the others in the sub-problem. So, still O(n)

17. **2 pts** Now that we have explored the effect of changing the c parameter as part of splitting problems into \([c, n-c]\), let’s see why merge sort (and binary search, arraylist growth, etc) does not split problems by subtracting constants. Write the recurrence equation for merge sort if it splits problems into \([c, n-c]\) at each level of recursion. Find the complexity bound (using the master theorem). Does changing the constant c affect the Big-O complexity?

**You can’t use the master theorem to solve this recurrence.**